使用Node.js开发多人玩的HTML 5游戏

《星噬》确实引起了我的兴趣，因为它很简单，但玩法很吸引人，不过明显缺少支持多人玩的功能。我一下子来了劲，想解决这个问题。于是，osMUs(MU指多人玩)应运而生，这是一款基于浏览器的多人玩的《星噬》克隆版游戏。

作者：布加迪编译 来源：51CTO | 2011-12-16 10:08

【51CTO精选译文】有一天，几个朋友来我家，给我介绍几个很酷的iPad游戏。其中一个游戏是《星噬》(Osmos)，开发这款游戏的是加拿大一家独立开发商，名叫Hemisphere Games。你可以控制在二维空间漂浮的一个小小的星团。小星团唯一能做的事就是往某个特定的方向喷射自己，结果往相反的方向推动星团。游戏规则很简单;主要规则就是，两个星团碰撞时，大的那个会吞噬掉小的那个。其余规则基本上直接来自质能守恒。

《星噬》确实引起了我的兴趣，因为它很简单，但玩法很吸引人，不过明显缺少支持多人玩的功能。我一下子来了劲，想解决这个问题。于是，osMUs(MU指多人玩)应运而生，这是

一款基于浏览器的多人玩的《星噬》克隆版游戏。

工作原理

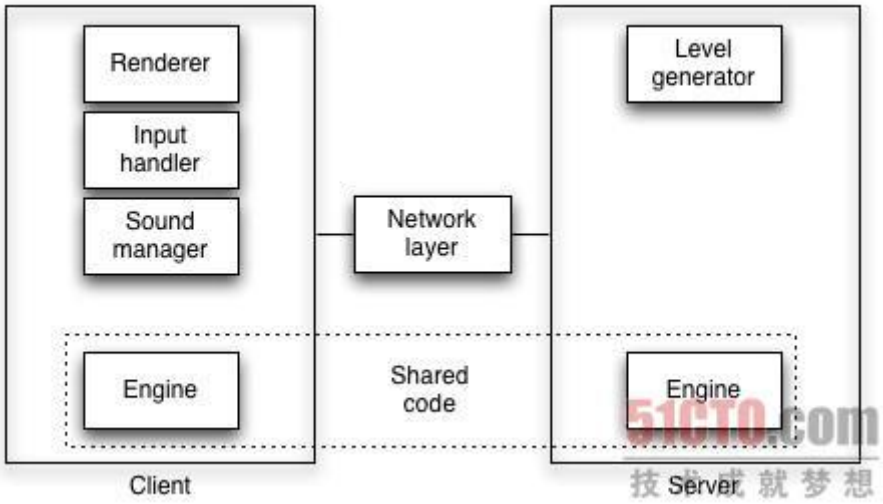
浏览器浏览到osmus登录页面后，服务器会将宇宙的当前状态发送给新的客户端，这个宇宙由多个速度随机的星团组成。这时候，客户端可以被动地关注游戏进度;但是当然了，也可以作为玩家控制的星团，加入游戏。一旦玩家加入，他就可以点击或在移动设备上快速按下画布(canvas)，射出新的星团。

随着游戏不断进行，服务器决定某人(可能是其中一个独立自主的星团)何时获胜;这时，玩家们接到通知，游戏重新开始。

本文其余部分介绍了与开发有关的一些具体内容。所以，如果你想试一下，尽管试好了。不过要注意一点：osmus在Chrome稳定版(版本13)和iPad上运行。

游戏架构

我编写osmus，是为了分成不同的、松散耦合的组件，既为了让其他代码贡献者更容易获得代码库，又为了便于尝试可以互换的技术。



osmus使用一个共享的游戏引擎(Game Engine)，该引擎既可以在浏览器中运行，又可以在服务器上运行。引擎是一个简单的状态机，其主要功能就是使用里面定义的物理规则，计算出与时间有关的下一个游戏状态。

```
1. Game.prototype.computeState = function(delta) {
2.   var newState = {};
3.   // Compute a bunch of stuff based on this.state
4.   return newState;
5. }
```

这是游戏引擎很狭窄的定义。在游戏开发领域，游戏引擎的含意通常涵盖渲染器、声音播放器和网络层等方面。这种情况下，我在这些组件之间作了非常明确的划分，osmus游戏的核心仅仅包括物理状态机，那样客户端和服务器都能计算出下一个状态，因而在时间上做到很合理的同步。

客户端有三个主要部件组成：渲染器、输入管理器和声音管理器。我制作了一个非常简单的基于画布的渲染器，将星团画成红圆圈，将玩家星团画成绿圆圈。我的同事Arne Roomann-Kurrik编写了一个替代的基于three.js的渲染器，使用了一些壮丽的着色器和阴影。

声音管理器处理回放声音效果和背景音乐(来自8-bit Magic)的工作。目前实现的方法使用了音频标签，有两个元素，一个用于背景音乐通道，另一个用于声音效果通道。这个方法存在已知的局限性，但考虑到我实现的方法具有模块性，声音实现方法可以换成使用其他API的方法，比如使用Chrome的Web Audio API。

最后，输入管理器负责处理鼠标事件，但是可以换成改而使用触摸操作的管理器，用于移动版本。在移动情况下，可能有必要使用CSS3转换而不是使用画布，因为CSS3在iOS上是硬件加速的，而HTML5画布仍然不是，也没有实现WebGL。

说到移动，我惊喜地发现，osmus在iPad上玩起来很顺畅，尤其是在运行最新iOS版本的iPad 2上。这太好了，也是为开放互联网编写游戏的其中一个实际好处。

联网很难

从联网的角度来看，游戏是一个相当宏伟庞大的项目，需要客户端之间实现无缝实时同步。正由于如此，客户端/服务器的双向通信必不可少。在现代互联网架构中，这种通信机制由Web Sockets来提供，它在TCP上提供了薄薄的一层，把许多繁琐的细节隐藏起来，不让实现者看到。为进一步隐藏网络堆栈方面的细节，我使用了socket.io库，该库为整个游戏提供了一种异常简单的事件驱动抽象层。遗憾的是，目前不支持二进制数据，不然可以大大压缩消息大小——拿《星噬》来说，压缩后也许可以减少一两个数量级。

经过一番研究，包括我与知名的HTML5开发专家Rob Hawkes进行的那次深入讨论后，清楚地发现：要获得任何一种共享体验，最简单的模式就是在服务器上有真正的游戏状态，让客户端定期与它进行同步。这方面需要取舍的主要是同步质量与所需的网络流量。

在一个极端情况下，如果游戏逻辑完全在服务器上，以每秒60帧的速度将更新内容(或者可能仅仅是屏幕截图)发送到客户端，就可以编写游戏，但是由于这种模式需要数量庞大的带宽，所以这个做法一般行不通。在相反的极端情况下，你可以设想这种网络架构：客户端连接，获得初始状态，然后基本上各自独立自主。

> 实际上，有一种很好的折衷方法——许多支持多人玩的游戏采用这种方法，那就意味着复制客户端和服务器中的重要代码。幸好，由于我们处在无所不在的JavaScript时代，再也不需要复制功能，而是只要用JavaScript编写游戏引擎就可以共享代码，然后在客户端上的浏览器中和服务器上的node.js中运行即可。
>
> **共享的JS模块**
>
> 如前所述，osmus使用在客户端与服务器之间共享的物理引擎。因而有人可能会想：在两者之间共享JavaScript代码会易如反掌，实际上不是那么容易。
>
> 模块加载器有一大堆。有CommonJS规范、RequireJS库和node.js require方法，没有一个可以很好地协同使用。如果你不用模块加载器，就想在客户端和服务器之间共享代码(这是服务器上JS的一大优点)，那么你可以使用这个有点变通的模式：

```
1. (function(exports) {
2.
3. var MyClass = function() { /* ... */ };
4. var myObject = {};
5.
```

```
6.  exports.MyClass = MyClass;
7.  exports.myObject = MyObject;
8.
9.  })(typeof global === "undefined" ? window : exports);
```

这个变通方法靠的是这一点：node.js定义了global(全局)对象，而浏览器没有定义。有了这个变通方法，node.js require()会很高兴，你还可以在<script>标签中加入文件，不会污染你的名称空间，当然假设没有其他JS以window.global对象污染你的名称空间！

遗憾的是，这个方法只适用于一个共享模块。一旦你有了多个彼此依赖的模块（通过node-land中的require方法和browser-land中的global对象），节点的名称空间与浏览器的加入之间的差异会变得异常明显，需要更多的变通方法。

另一个方法是使用browserify，捆绑所有JS，在浏览器里面模拟require。这种方法依赖node.js来提供生成的JS，这并不理想，因为静态文件应该由专门为该用途优化的web服务器来提供。不过，node.js+ browserify可以进行配置，以便编译可以静态提供的JS，不必依赖节点来提供。这个方法带来了一些开销：

1. 多出了构建这个步骤，以便部署。

2. 无论browserify使用什么机制来支持require()调用，都需要性能开销。

总的来说，这个方法在我看来比较好，我希望在将来编写的osmus版本中试用一下。

原文：Developing Multiplayer HTML5 Games with Node.js

【编辑推荐】

1. 你应该知道的Node.js扩展模块——Hashish

2. Node.js提速指南
3. 关于Node.js语言的讨论
4. Node.js初体验
5. 什么是Node.js？

【责任编辑：陈贻新 TEL：（010）68476606】

点赞 0

Node.js 分享:

大家都在看 猜你喜欢